# Bahria University,

## Karachi Campus



LAB EXPERIMENT NO.

**\_03\_**

LIST OF TASKS

|  |  |
| --- | --- |
| **TASK NO** | **OBJECTIVE** |
| 01 | Implement BFS & DFS Algorithm in python on the given graph: |
| 02 | Implement the BFS and DFS Algorithm using recursion on the given graph with starting node = 1 and goal =6 |
| 03 | Apply the UCS algorithm on a map given below. Find optimal cost from ARAD to BUCHAREST |
| 04 | Implement the Travelling Salesmen problem using uninformed searches on given Directed graph |
|  |  |

Submitted On:

Date: 27/02/2024

**Task No 01:** Implement BFS & DFS Algorithm in python on the given graph:

**Solution:**

def bfs(graph, start):

    visited = set()

    queue = [start]

    bfs\_traversal = []

    while queue:

        vertex = queue.pop(0)

        if vertex not in visited:

            bfs\_traversal.append(vertex)

            visited.add(vertex)

            queue.extend(graph[vertex])

    return bfs\_traversal

def dfs(graph, start):

    visited = set()

    stack = [start]

    dfs\_traversal = []

    while stack:

        vertex = stack.pop()

        if vertex not in visited:

            dfs\_traversal.append(vertex)

            visited.add(vertex)

            stack.extend(reversed(graph[vertex]))

    return dfs\_traversal

graph = {

    'A': ['B', 'C','D'],

    'B': ['E', 'F'],

    'C': ['F'],

    'D': [],

    'E': [],

    'F': []

}

start\_node = 'A'

bfs\_result = bfs(graph, start\_node)

dfs\_result = dfs(graph, start\_node)

print("BFS Traversal:", bfs\_result)

print("DFS Traversal:", dfs\_result)

**Output:**

A group of letters on a white background

Description automatically generated

**Task No 02:** Implement the BFS and DFS Algorithm using recursion on the given graph with starting node = 1 and goal =6

**Solution:**

def bfs(graph, start, goal, queue=None, visited=None):

    if queue is None:

        queue = [start]

    if visited is None:

        visited = set()

    if not queue:

        return []

    vertex = queue.pop(0)

    visited.add(vertex)

    if vertex == goal:

        return [vertex]

    for neighbor in graph[vertex]:

        if neighbor not in visited and neighbor not in queue:

            queue.append(neighbor)

    return [vertex] + bfs(graph, start, goal, queue, visited)

def dfs(graph, current, goal, visited=None):

    if visited is None:

        visited = set()

    visited.add(current)

    if current == goal:

        return [current]

    for neighbor in graph[current]:

        if neighbor not in visited:

            path = dfs(graph, neighbor, goal, visited)

            if path:

                return [current] + path

    return []

graph = {

    1: [2, 3,4],

    2: [5,6],

    3: [6],

    4: [7,8],

    5: [9,10],

    6: [],

    7:[11,10],

    8:[],

    9:[],

    10:[],

    11:[],

}

start\_node = 1

goal\_node = 6

bfs\_result = bfs(graph, start\_node, goal\_node)

dfs\_result = dfs(graph, start\_node, goal\_node)

print("BFS Path:", bfs\_result)

print("DFS Path:", dfs\_result)

**Output:**

**A number with black numbers

Description automatically generated with medium confidence**

**Task No 03:** Apply the UCS algorithm on a map given below. Find optimal cost from ARAD to BUCHAREST

**Solution:**

import heapq

graph = {

    'Arad': [('Zerind', 75), ('Sibiu', 140), ('Timisoara', 118)],

    'Zerind': [('Arad', 75), ('Oradea', 71)],

    'Oradea': [('Zerind', 71), ('Sibiu', 151)],

    'Sibiu': [('Arad', 140), ('Oradea', 151), ('Fagaras', 99), ('Rimnicu Vilcea', 80)],

    'Timisoara': [('Arad', 118), ('Lugoj', 111)],

    'Lugoj': [('Timisoara', 111), ('Mehadia', 70)],

    'Mehadia': [('Lugoj', 70), ('Drobeta', 75)],

    'Drobeta': [('Mehadia', 75), ('Craiova', 120)],

    'Craiova': [('Drobeta', 120), ('Rimnicu Vilcea', 146), ('Pitesti', 138)],

    'Rimnicu Vilcea': [('Sibiu', 80), ('Craiova', 146), ('Pitesti', 97)],

    'Fagaras': [('Sibiu', 99), ('Bucharest', 211)],

    'Pitesti': [('Rimnicu Vilcea', 97), ('Craiova', 138), ('Bucharest', 101)],

    'Bucharest': [('Fagaras', 211), ('Pitesti', 101), ('Giurgiu', 90), ('Urziceni', 85)],

    'Giurgiu': [('Bucharest', 90)],

    'Urziceni': [('Bucharest', 85), ('Hirsova', 98), ('Vaslui', 142)],

    'Hirsova': [('Urziceni', 98), ('Eforie', 86)],

    'Eforie': [('Hirsova', 86)],

    'Vaslui': [('Urziceni', 142), ('Iasi', 92)],

    'Iasi': [('Vaslui', 92), ('Neamt', 87)],

    'Neamt': [('Iasi', 87)]

}

def ucs(graph, start, goal):

    frontier = [(0, start)]

    explored = {start: 0}

    while frontier:

        current\_cost, current\_node = heapq.heappop(frontier)

        if current\_node == goal:

            return explored[current\_node]

        for neighbor, cost in graph[current\_node]:

            total\_cost = current\_cost + cost

            if neighbor not in explored or total\_cost < explored[neighbor]:

                explored[neighbor] = total\_cost

                heapq.heappush(frontier, (total\_cost, neighbor))

start\_city = 'Arad'

goal\_city = 'Bucharest'

optimal\_cost = ucs(graph, start\_city, goal\_city)

print("Optimal cost from", start\_city, "to", goal\_city, ":", optimal\_cost)

**Output:**

****

**Task No 04:** Implement the Travelling Salesmen problem using uninformed searches on given Directed graph

**Solution:**

graph = [

    [0, 10, 15, 20],

    [10, 0, 35, 25],

    [15, 35, 0, 30],

    [20, 25, 30, 0]

]

def dfs\_tsp(graph, start, visited, path\_length):

    if len(visited) == len(graph):

        return path\_length + graph[start][0]

    min\_cost = float('inf')

    for city in range(len(graph)):

        if city not in visited and city != start:

            visited.add(city)

            current\_cost = dfs\_tsp(graph, city, visited, path\_length + graph[start][city])

            min\_cost = min(min\_cost, current\_cost)

            visited.remove(city)

    return min\_cost

def solve\_tsp\_dfs(graph):

    start\_city = 0

    visited = {start\_city}

    path\_length = 0

    optimal\_cost = dfs\_tsp(graph, start\_city, visited, path\_length)

    return optimal\_cost

optimal\_cost = solve\_tsp\_dfs(graph)

print("Optimal cost for TSP using DFS:", optimal\_cost)

**Output:**

**A black text on a white background

Description automatically generated**